Cover Feature

Design and Assurance Strategy for the NRL Pump

The NRL Pump forwards messages from a low-level system to a high-level system and monitors the timing of acknowledgments from the high system to minimize leaks. It is the keystone to a proposed architecture that uses specialized high-assurance devices to separate data at different security levels.

Myong H. Kang
Andrew P. Moore
Ira S. Moskowitz
US Naval Research Laboratory

In the past 20 years, only a handful of high-assurance, multilevel, secure computers have been built, and even these are rarely used in operational environments. Such systems suffer a host of disadvantages: they cost too much, lack user-friendly features and development environments, take too much time to evaluate and certify, and do not scale well for secure distributed computing. The sidebar "How Multilevel Secure Computing Works" explains the requirements for multilevel secure computing.

This lack of satisfactory security solutions is disturbing in light of the trend toward open and distributed computing, which increases a system's vulnerability to attack. We believe it is vital to develop scalable security solutions that do not depend solely on current architectures for multilevel secure computing. We propose basing security solutions instead on a *multiple single-level security architecture*,[1,2] which uses commercial (nonsecure) products for general-purpose computing and special-purpose high-assurance devices to separate data at different security levels. A multiple single-level architecture is a viable and practical solution to distributed multilevel secure computing, as described in the sidebar. It can monitor interactions in a way that minimizes information leaks from a high security level to a lower security level. Also, it uses only a handful of trusted devices to separate information, which means a less expensive and shorter evaluation and certification process.

The keystone of this architecture is a trusted device that "pumps" data from a low security level to a higher one. In this article, we describe the software design and assurance argument strategy for this device, the Network NRL Pump, which can be used in any multilevel secure distributed architecture. We have completed the system requirements and logical design of a prototype pump and are working on its physical design. We plan to complete the prototype and its related assurance argument some time in 1999.

DESIGN OVERVIEW

In 1993, Kang and Moskowitz introduced the basic NRL Pump,[3] which they and Daniel Lee later[4] extended to a network environment. The Network NRL Pump is the version of the NRL Pump we describe in this article.

Figure 1 shows the pump with wrapper and application software. Messages are sent from a *low system*—a system operating at a low security level—to a *high system*—a system operating at a high security level—but not in the reverse direction. The pump and its software work in the following manner:

- The low system sends a message to the high system through the pump.
- The pump stores it in its buffer and sends an acknowledgment to the low system so that the low system knows its message was received. The timing of this acknowledgment is probabilistically based on a moving average of acknowledgment times from the high system to the pump.
- The low system cannot send any new messages until it gets an acknowledgment of previous messages sent.
- The pump stores the message until the high system is ready to receive it and then the pump forwards the message to the high system.
- The high system sends an acknowledgment to the pump.

How Multilevel Secure Computing Works

The controlled sharing of information is vital to the operation of most modern computing systems. Mandatory controls have a basis in law or organization-wide policy. While typical users are required to obey mandatory controls, often there are other controls that leave the user with some choice (discretion) over whether or not to disclose data to others. The military system for protecting sensitive information is a system of mandatory controls. Sensitive information is classified into levels according to the degree of damage its disclosure could cause to national security, and users are assigned appropriate clearances. Users are permitted to view a particular level of sensitive data only if their clearance qualifies them (and if they have a job-related need to view that data).

A computer system may store and process information with a range of classification levels and provide service to users with a range of clearances. If some users lack clearance for some of the information it holds, the system is said to be *multilevel secure*.

Multilevel secure computing lets users access information classified at or below their clearance, but prevents them from obtaining access to information classified above their clearance. A common way to enforce this is through a policy similar to "no reading upper level information and no writing down to a lower level."

In distributed computing, you can achieve a system-wide multilevel secure policy by sparsely using trusted *multilevel secure components* to hook up *single-level systems* at different security levels, thus creating a *multiple single-level security* architecture. The NRL Pump, described in the main text, lets information be securely sent from a system at a lower security level to one at a higher security level. In a generalized system that wishes to enforce a multilevel security policy, sanitized information (information processed at the high level) may need to be sent to the lower level. For this, a *downgrader*, a specialized trusted component, can be used to minimize leaks of high information to lower level systems. Specialized multilevel secure workstations could also be used to let a high-level user access lower level resources, for example, through the Web. These three specialized components make up the backbone of the multiple single-level security architecture.

Acknowledgments are necessary for reliable communication. If the high system passed acknowledgments directly to the low system, there would be a security problem because by altering the acknowledgment timing, the high system could encode an illicit message.[5] Such covert channels are dangerous because even if all other means of communication are cut off, an exploiter can still use such a sneaky method to violate the security requirement.[4] For this reason, the pump decouples the acknowledgment stream. However, for performance reasons, the long-term high-system-to-pump behavior should be reflected in the long-term low-system-to-pump behavior. For this reason, the pump uses statistically modulated acknowledgments.

As the figure shows, the pump is configured as a single hardware box with interfaces to the low system's LAN, (*low LAN*), the high system's LAN (*high LAN*) and an *administrator workstation*. The pump administrator, who is cleared for high-level data, uses the workstation to load configuration information into the pump (described in more detail later) and monitor its operation as necessary. The pump supports a specialized protocol, the *pump protocol*, across the LAN interfaces.

Within the low and high systems are *wrappers*, software that supports a variety of applications. Wrappers, which run on the low and high applications,

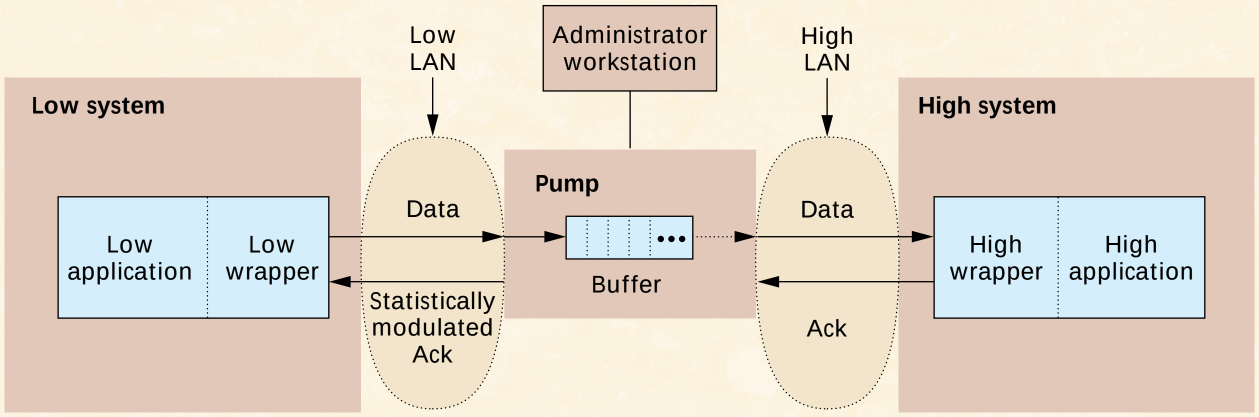

Figure 1. The NRL Network Pump with wrappers and applications. The pump minimizes the ability of a system operating at high security to influence acknowledgment timing. This reduces the threat of a covert communication channel being established from a high-security system to a low-security system.

Developing a trustworthy system is not easy because the developers must construct an assurance argument and have it evaluated by an independent certification team.

communicate with the pump over their respective LANs. Although not shown in the figure, each wrapper consists of two parts. The *application-specific* part provides an application-specific protocol on one side and the pump's API on the other. It can be tailored to support the particular set of objects or calls to the application it expects to see. The *pump-specific* part is a library of routines that implement the pump protocol. It supports the pump's API on one side and the pump protocol on the other. The application-dependent routines can call the pump-specific routines as required.

This structure has several interesting aspects:

- Only application programs that can operate with very little information returned to the sender from the receiver (for example, applications that use asynchronous communication) can use the pump.
- The pump's confidentiality properties depend solely on the pump itself, not on the wrappers. Thus, wrapper software is not security-critical and can be altered or replaced without affecting system security.
- The low wrapper is a proxy of the high application. It receives messages from the low application, delivers them to the pump, receives acknowledgments from the pump, and generates application-specific acknowledgments. Occasionally, one application message from a low application may be transformed into several pump messages.
- The high wrapper is a proxy of the low application. It receives a message from the pump, delivers it to the high application, receives application-specific acknowledgments, and converts them to pump-specific acknowledgments.

ASSURANCE ARGUMENT

Information systems that counter security threats must isolate the system's security-critical function into simple, well-defined, reusable components that can be trusted to carry out the security-critical function. A detailed explanation, or *assurance argument*, describes why this isolation is effective and why the critical components are trustworthy. A system or component is trustworthy if there is an acceptably high probability that it satisfies all its critical requirements.

Developing a trustworthy system is not easy because the developers must construct an assurance argument and have it evaluated by an independent certification team. The argument must instill high confidence that the system does what it is supposed to do—and only that. A convincing argument thus requires that the development process be understandable and that the implementation clearly conform to the critical requirements.

The judicious use of formal methods can strengthen a system's assurance argument because the tools of mathematics and logic ensure that critical properties hold. However, increasing the formality of an argument does not necessarily make it more convincing to an independent certifier unfamiliar with these tools. Constructing a persuasive and cost-effective argument often requires using many languages, methods, and tools—both formal and informal. Developers must present formal specifications and analyses in the context of the overall assurance argument or much of their persuasive power may be lost.[6]

Figure 2 illustrates our strategy for constructing the pump's assurance argument, in which we integrate formal specifications and analyses with structured system documentation. Along the left side are the primary levels of system refinement and documentation. Along the right side are the specification languages and tools used for implementation, analysis, and verification: I-Logix's Statemate graphical specification and simulation tools,[7] ORA Canada's Verdi/EVES formal verification environment,[8] and Reliable Software Technology's Whitebox DeepCover tools for analyzing test coverage.[9] The result of integrating the two sides (the center area between the two sets of arrows) is the pump's assurance argument. The top-down widening of the argument reflects the additional detail specified at the lower levels. Red lines show completed work; green lines show work in progress, and blue lines show work planned.

A variety of formal and informal techniques allow reasoning across five semantic domains: English narrative, Statemate logical design, Statemate physical design, formal Verdi PDL specification, and C++ code.

The network's interconnection requirements are expressed in English. The pump's critical requirements are also expressed in English using the primitives of a logical view, which is specified graphically in Statemate activity charts. We refine this logical view using a combination of Statemate activity and state charts, which constitutes the pump's behavioral view. This refinement consists of mapping the logical design components to a physical architecture, which is described in Statemate module charts.

The physical architecture is in turn mapped to a formal Verdi specification of each module's access program (interface function) requirements. The developer must show that the implementation conforms to the Verdi specification, either through formal proof, using the EVES verification system, or through testing, using the Whitebox DeepCover tool for coverage analysis. The type of verification performed depends on the complexity and type of requirement (functional, security, performance, and so on) and the code's complexity.

We have studied the behavior and vulnerability of the pump algorithms during normal operation,[4] but not dur-

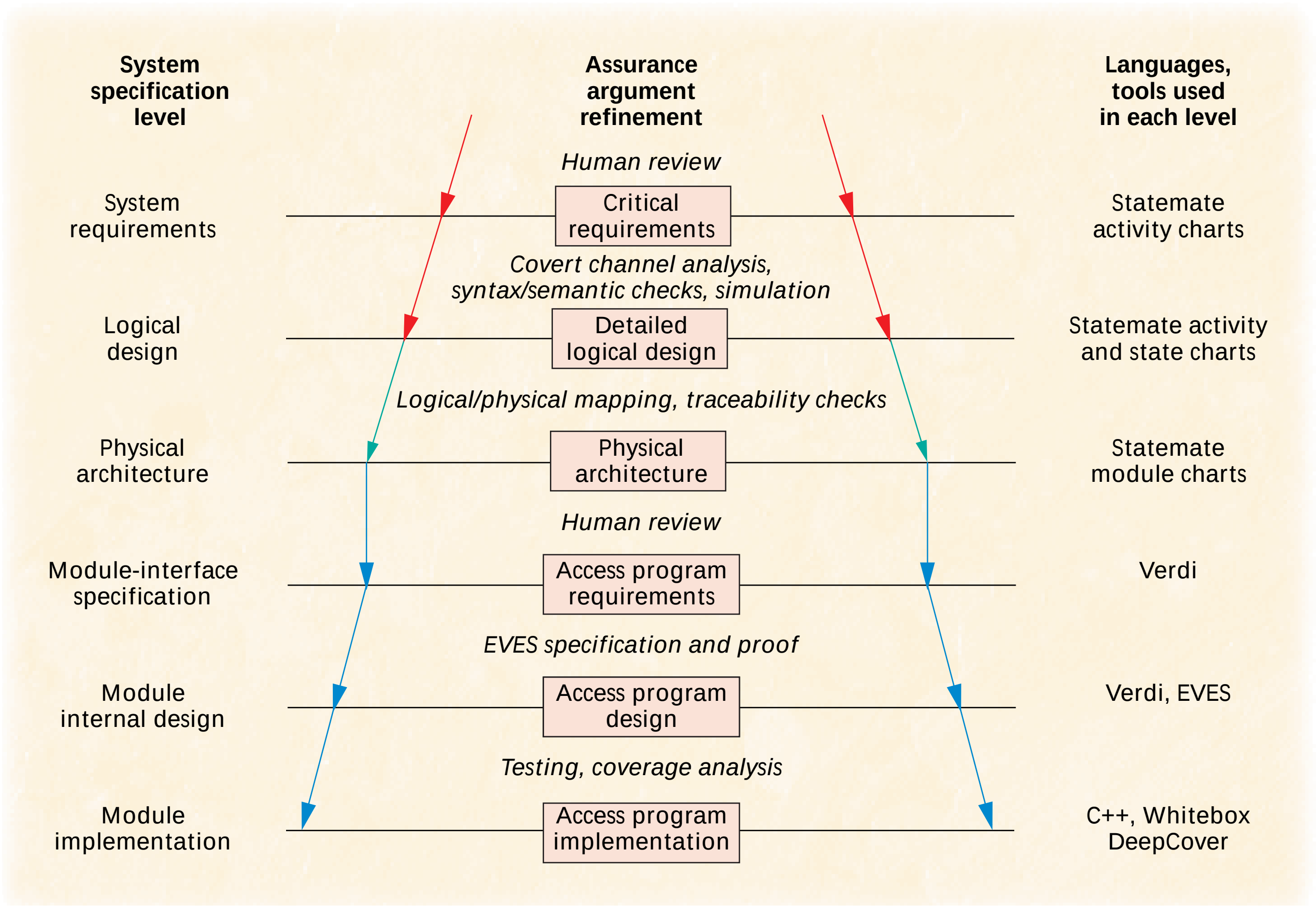
System specification level
Assurance argument refinement
Languages, tools used in each level
Human review
System requirements
Critical requirements
Statemate activity charts
Covert channel analysis, syntax/semantic checks, simulation
Logical design
Detailed logical design
Statemate activity and state charts
Logical/physical mapping, traceability checks
Physical architecture
Physical architecture
Statemate module charts
Human review
Module-interface specification
Access program requirements
Verdi
EVES specification and proof
Module internal design
Access program design
Verdi, EVES
Testing, coverage analysis
Module implementation
Access program implementation
C++, Whitebox DeepCover

Figure 2. Strategy for building the pump's assurance argument—the case for why the device is considered trustworthy. On the left side are the stages of the pump's design. The widening of the center region indicates a refinement of the design specification to a more detailed specification or implementation; vertical arrows indicate a translation of a specification from one semantic domain to another at a comparable abstraction level. The center region forms the assurance argument. The increased width from the top down reflects the additional detail specified at the lower levels. Red lines show work completed to date, green lines show work ongoing, and blue lines show work planned.

ing connection initialization, error handling, or error recovery. We are developing a fully functional prototype that reflects the pump's logical design to study vulnerabilities in the operational environment, such as an unanticipated overuse of pump resources. We expect studies of the prototype to clarify the pump's administrative requirements (such as the initialization, procedure to establish connections) and its monitoring activities, as well as error handling and audit requirements.

As we mentioned earlier, an assurance argument requires easy-to-understand mappings from the critical system requirements to the design. To support this requirement, the prototype design has three main goals:

- Clearly define each module's task (module being the thread or object in the prototype).
- Separate modules that interact with the low system from modules that interact with the high system and label them low and high, respectively.
- Reduce the communication between low modules and high modules as much as possible to simplify monitoring and security verification.

System requirements

Because the prototype is not complete, we can describe only the first two levels of the assurance argument in Figure 2. The first level is the pump's system-level requirements. In this view, we describe the pump as an outsider would see it, with the pump itself being a black box.

Configuration and administration. In a simplistic view, the pump is just a network router that connects a low network to a high network. However, it cannot accept a message from a process at a lower security level and route that message to just any process at a higher security level. Allowing such uncontrolled

To make connection recovery easier, the pump operates at the application layer and communicates to the low and high systems through pump messages.

behavior would mean that any low process could establish a connection to any high process, which would waste pump resources. Moreover, a low Trojan Horse process could ping high processes to see if any high Trojan Horse processes exist. The pump should prevent such arbitrary exploitation, but at the same time be available to many applications.

To meet these requirements, processes that use the pump must register their addresses with the pump administrator. The pump administrator, who verifies that the registration is legitimate, can enter the addresses of registered processes into the pump's *configuration file*, which the pump administrator manages. Because the pump checks the configuration file only when a connection is established, the administrator can change the file and reload it at any point during the pump's normal operation.

The file contains

- *Pump initialization information*. This includes things like window size, maximum number of connections, and the length of time the pump must maintain the last message if the connection abnormally aborts.
- *Registered low and high processes.* These processes are likely to be the low and high wrappers because the pump typically communicates with COTS applications only through these wrappers. When the wrapper is registered, it also identifies the application it interacts with as being either recoverable or nonrecoverable (described later).
- *Allowable connections*. This information, which the pump administrator specifies in the connection table, is used to control network access when a low process sends a connection request to the pump.

The pump also has an external interface, which the administrator uses to load configuration files, request pump status, and so on. When the administrator requests the pump's status, the interface returns the status of active and aborted connections (how long the connection was active or idle). When the connection's idle time is too long, the pump administrator can kill the connection.

Finally, the pump maintains a well-known port to which a low process can send a connection request to a specific high process.

Recovery. Once a low wrapper receives an acknowledgment from the pump, it must be able to safely assume that the pump will deliver the message to the corresponding high wrapper, even if there is a power failure or system crash in either the pump or high wrapper.

Applications will have either a nonrecoverable or recoverable connection, which the wrapper identifies. The type of connection determines how the pump will act if the connection is abnormally aborted. An FTP client and server, for example, are *nonrecoverable*; that is, if the connection breaks in the middle of a file transfer, they will not expect the file transfer to be recovered when the connection resumes. However, a connection between a Sybase replication server and an SQL server is *recoverable*: if the connection abnormally breaks, they expect recovery after the connection is resumed.

Different recoverable applications have different recovery procedures, so their wrappers must maintain the information necessary for recovery. In the Sybase replication server and SQL server exchange, for example, the wrapper must keep the last message the replication server sends to the SQL server, because it is for synchronization. However, because the wrapper cannot predict when the connection will be aborted, it must write every "last" message to persistent storage. In general, the low and high systems in which the wrappers reside are nonrecoverable, and maintaining persistent messages is usually very expensive, since every message must be written to disk and synchronized.

The pump itself is recoverable, however, which reduces the cost of maintaining an extra persistent message for a recoverable connection. If a recoverable connection is aborted, the pump maintains the last message it receives from the low system, even if it has delivered all the other messages to the high system. However, because it cannot keep it indefinitely, the pump maintains this last message for only T hours, a parameter specified in the configuration file. The pump administrator can always reclaim the resources from the aborted connection after T hours.[10]

Message classes and procedure to establish connections. To make connection recovery easier, the pump operates at the application layer and communicates to the low and high systems through *pump messages*. The data message and control message classes are inherited from the generic pump message class. (Technically, these message classes are part of the logical design in the next level, but because they are directly concerned with how outsiders interact with the pump, we decided to describe them here.)

The procedure to establish connection involves exchanging control messages among the low system, high system, and pump. When the low system sends a *connection request* message to the pump, it identifies itself with its own address and type of application (recoverable or nonrecoverable). It also specifies the address it wishes to connect to.

The pump will check the configuration file to determine if the request is permitted. If the low system and high system addresses match addresses in the connection table, the pump sends a *connection valid* message to the low system. If the connection request originated from an unregistered low process, the pump ignores the request. If a registered low process requests a connection not specified in the connection table (an illegal request), the pump sends a *connection rejected* message to the low system. When the low system receives a connection valid message, it disconnects the current connection and is ready to accept a new connection from the pump. This redundant connection procedure acts to verify the address of the low system. The low system then uses the new connection to transmit data.

Registered high processes are always ready to accept a connection from the pump. Once the pump validates the connection request from the low system, it initiates a new connection to the high system by relaying the connection request message that came from the low system. The high system validates the request and sends a connection valid or connection rejected message to the pump. When the new connection is established, the pump sends a *connection grant* message with initialization parameters to the high system. If the connection is recoverable and the previous connection was abnormally disconnected, the pump will send an *undelivered* message from the previous session to the high system. If the connection to the high system is successfully established and all undelivered messages are cleared, the pump establishes a connection to the low system and sends a connection grant message to the low system. If the connection is recoverable and the previous connection was abnormally disconnected, the pump sends the last data message it received from the low system for synchronization. If the pump cannot establish the connection to the high system or clear all undelivered messages from the previous session, it establishes a connection to the low system and sends it a *connnection exit* message. The pump also sends connection exit messages to both the low and high systems when it is ready to shut down the connection because of error or the administrator's request.

Once the pump establishes a connection from the low system to the high system, it uses the data message to send information. *Ack* is a special data message with zero data length (the first two bytes are zero) that can be sent from the high system to the pump and from the pump to the low system. Another special data message is *connection close*, which is sent at the end of normal data transmission. The message, which requests a normal connection close, can be sent from the low system to the pump and from the pump to the high system. The connection close message would seem to be a control message, not a data message. However, it must propagate from the low system to the high system through the pump in sequence. To make it a control message, we would have to introduce an extra communication channel from the low system to the high system. By sending it as a data message through the established connection, not only do we avoid the need for an out-of-band signal, we are also assured that the connection close message will be processed in the correct order. That is, by the time the high system receives the message, all other data messages should have processed. In general, the connection close message originates from low wrappers, not low applications.

Once the pump establishes a connection from the low system to the high system, it uses the data message to send information.

Logical design

The second level of the assurance argument in Figure 2 is logical design. This internal view of the pump must support the first level of the assurance argument and the three design goals given earlier, especially mapping functions from system requirements to parts of the logical design.

The pump has three data structures: the connection table (one per pump), the connection buffer (one per active or aborted connection), and the pump messages just described. It also has three types of threads: the main thread, trusted low threads, and trusted high threads. A significant challenge in logical design is error handling.

Connection table. The pump maintains a connection table that records the status of all active and aborted connections. If there is a legal connection in the configuration file, there is one entry in the connection table; if the connection is neither active nor aborted, there is no entry in the connection table. Each entry records the connection's status (active or aborted), the address of its connection buffer, the addresses of the high and low pointers to the trusted high threads and trusted low threads (null if the connection is inactive), and the time of the last activities of either the trusted high threads or trusted low threads.

Connection buffer. Each connection between a low sender and a high receiver has one FIFO-bounded buffer controlled by a monitor and two threads: the trusted low thread puts messages in the buffer, while the trusted high thread removes them. The connection buffer stores an array of handles of data messages and a variable that records the moving average of the outgoing message rate (the trusted high thread's consumption rate), which the trusted low thread uses to control the stochastic delay for acknowledgments to the low system.

The pump creates a connection buffer when the low system requests a new, valid connection to a high process *and* if there is no preexisting connection buffer from an aborted connection between the same pair of low and high ports. A connection buffer is deleted when a connection terminates normally (with a connection close message).

Threads. The *main thread* initializes the pump. This includes reading the configuration file to track relevant information for each connection. The main thread also "listens" to the well-published port of the pump to

The pump is a secure one-way communication device that minimizes any direct or indirect communication from the high system to the low system.

which the low system sends connection request messages. In response to a valid request, the main thread spawns a connection that consists of a trusted high thread, a trusted low thread, and a connection buffer. It then sends a connection valid or connection rejected message to the low system, depending on the validity of the request (or it ignores the request if the request is from an unregistered low process). The main thread also populates the connection table as it spawns a connection, providing the connection's status and pointers to the trusted high thread, trusted low thread, and connection buffer. The rest of the connection setup is done by both the trusted high thread and the trusted low thread. After exchanging necessary control messages, the low system starts sending data messages.

The *trusted high thread* establishes a connection to the high system by sending a connection request message. After it receives a connection valid message, the trusted high thread sends a connection grant message that contains the connection ID, maximum message size, and so on. The trusted high thread then delivers any leftover data messages in the buffer from the previous (aborted) session to the high system. When the buffer is empty, it awakens the trusted low thread. The trusted high thread keeps delivering messages as long as there are messages in the connection buffer. It also updates the moving average according to acknowledgment times from the high system.

Both the trusted high thread and trusted low thread use a sliding window scheme. The window size *w* is specified in the configuration file (as part of the pump initialization information). In the scheme, a trusted high thread can send up to *w* data messages from the buffer without receiving acknowledgments from the high system. The pump protocol requires the high system to send acknowledgments to the trusted high thread in the same order it receives messages. If the high system violates pump protocol by sending an out-of-sequence acknowledgment, the trusted high thread sends a connection exit message to the high system, disconnects the pump from the high system, and logs the high system's misbehavior.

When a *trusted low thread* is created, it waits for a trusted high thread to awaken it. This occurs when all undelivered messages from the previous session are delivered to the high system if the connection is recoverable. The trusted low thread then establishes a connection to the low system and sends it a connection grant message. If the application is recoverable and the last message is not a connection close message received from the previous session (there was an abnormal disconnection), the trusted low thread also sends the last message it received from the low system. It then starts to receive data messages from the low system. When it receives a data message, it verifies the message ID and connection ID, allocates memory, and stores the message's handle in the connection buffer. The trusted low thread also computes a statistical moving average, as described earlier, which is based on the moving average rate at which the trusted high thread consumes messages.

The trusted low thread receives up to *w* data messages without sending any acknowledgments to the low system. The trusted low thread must acknowledge messages in the same order it receives them from the low system, despite the probabilistic delay. To maintain the order and timing of the delayed acknowledgments, the trusted low thread maintains the current acknowledgment ID and the pending acknowledgment queue. When the trusted low thread computes the delay, it stores time values in ascending order that tell when to send the next acknowledgment in the queue. As soon as it reaches the first time value in the queue, it sends an acknowledgment with the current acknowledgment ID and increments the current acknowledgment ID.

Error handling. Error or failure handling is one of the most difficult parts of design because there is no theory or best way to handle errors. One important question is "How smart should the pump be for error recovery?" The smarter the pump, the more complex the software, and the harder it will be to ensure its correct behavior. We designed the pump's error handling with this in mind.[10]

SECURITY ANALYSIS

As Figure 2 shows, analyses performed on these specifications include a covert channel analysis, a Statemate analysis of their logical consistency and completeness (syntax/semantic checks), and Statemate simulations. We have specified the logical design just described in the Statemate tool set, and it has passed the correctness/completeness checks Statemate requires. Of course, this does not ensure that the design conforms to the pump's critical requirements. As the figure also shows, this requires a human review, a detailed covert channel analysis, and simulation of the logical design. We are currently soliciting comments on our design and have just started detailed conformance testing using the Statemate simulator. The rest of this section is devoted to the security of the logical design, including its covert channels.

As we have shown, the pump is a secure one-way communication device that minimizes any direct or indirect communication from the high system to the low system. Only the trusted high thread talks to the high system; the main thread and trusted low thread talk to the low system—and the main thread only during connection setup. Thus, all interaction involves only the trusted high thread and trusted low thread.

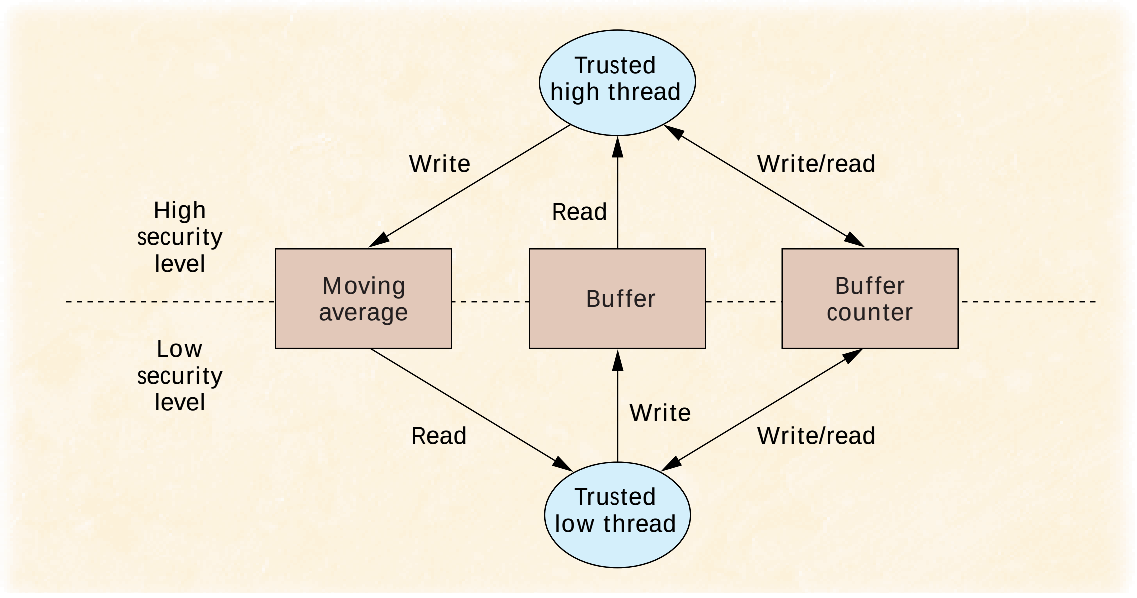

Figure 3. Pump design in terms of thread interactions. The main thread is not shown because it talks to the low system only when the connection is set up. The design avoids direct communication from the high security level to the low security level, and has only three indirect communication paths between the trusted high thread and trusted low thread. Of those, only the two outside paths must be monitored because the center path is moving only from the low to high level.

Even though these two threads are trusted software, however, we still minimize their interaction from high to low because any communication in that direction must be carefully monitored.

Figure 3 shows the pump design in the context of thread interactions during normal operation. There is no direct communication between the two trusted threads and only three indirect communication paths between them. Only two of these three paths must be monitored, however, because the middle path is one-way upward. These two paths and their effects on the low system are described in detail elsewhere.[3,4]

A major part of the pump's security design is its ability to mitigate covert timing channels from the high system to the low system. However, some information can still be sent in that direction because the pump notifies the low system when a connection is down and someone can manipulate the recovery processes to leak data.

When designing a secure device with any sort of realistic functionality, the best you can hope for is to *minimize* covert communication from the high system to the low system;[5] it is impossible to eliminate it. The pump design enforces a minimum time τ between connection reestablishment and the auditing of any connections that abort often. Thus, we have at worst introduced an additional covert channel with a capacity of approximately [number of connections] bits per τ. Furthermore, the audit process will easily detect any covert channel that attempts to send meaningful amounts of information using a disconnect/connect strategy.

Finally, in this analysis, we assume that the processes on the same level do not communicate among themselves. We plan to relax this condition in future work. Our preliminary research indicates that intra-LAN communication does not significantly increase covert channel capacity.

The NRL Network Pump provides a solid foundation for progress toward the proposed multiple single-level architecture. We are currently experimenting with mapping the three threads onto three processors. One processor would handle connection requests from the low system and communication to the administrator (main thread is mapped to this processor), one processor would handle all other communication to the low system (trusted low thread), and the last processor would handle all communication to the high system (trusted high thread). If we use only two processors, one processor could handle all communication to the low system (connection requests and data from the low system); the other processor could handle all communication to the high system and the administrator. ❖

References

1. M. Kang, J. Froscher, and I. Moskowitz, "A Framework for MLS Interoperability," *Proc. High-Assurance Systems Eng. Workshop*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 198-205.
2. M. Kang, J. Froscher, and I. Moskowitz, "An Architecture for Multilevel Secure Interoperability," *Proc. Computer Security Applications Conf.*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 194-204.
3. M. Kang and I. Moskowitz, "A Pump for Rapid, Reliable, Secure Communication," *Proc. ACM Conf. Computer and Comm. Security*, ACM Press, New York, 1993, pp. 119-129.
4. M. Kang, I. Moskowitz, and D. Lee, "A Network Pump," *IEEE Trans. Software Eng.*, May 1996, pp. 329-338.
5. I. Moskowitz and M. Kang, "Covert Channels—Here to Stay?" *Proc. Computer Assurance Conf.*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 235-243.

6. A. Moore and C. Payne, "Increasing Assurance with Literate Programming Techniques," *Proc. Computer Assurance Conf.*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 187-198.
7. D. Harel et al., "Statemate: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Software Eng.*, Apr. 1990, pp. 403-414.
8. S. Kromodimoeljo et al., "EVES: An Overview," Tech. Report CP-91-5402-43, ORA Canada, Ottawa, Ontario, 1993.
9. *Whitebox DeepCover: User Reference Manual,* Reliable Software Technologies, Sterling, Va., 1996.
10. M. Kang, A. Moore, and I. Moskowitz, "Design and Assurance Strategy for the NRL Pump," NRL Memo 5540-97-7991, Naval Research Laboratory, Washington, D.C., Dec. 1997.

Myong H. Kang is a computer engineer at the US Naval Research Laboratory, where he is designing a multilevel secure workflow and applying it to mission-critical applications. In previous work at NRL, he developed the prototype of Sintra multilevel secure database systems and related theories. His research interests include distributed computing, parallelization techniques, data modeling, consistency maintenance, and performance modeling in parallel and distributed systems. Kang received an MS from the University of Illinois at Urbana-Champaign and a PhD from Purdue University, both in electrical engineering.

Andrew P. Moore is a computer scientist at the Naval Research Laboratory's Center for High Assurance Computer Systems, where he develops methods and tools to more easily engineer affordable high-assurance systems and products. His research interests include high-assurance system development, software/hardware codesign, formal methods, and model checking. Moore received a BA in mathematics and computer science from the College of Wooster and an MA in computer science from Duke University.

Ira S. Moskowitz is a mathematician at the Naval Research Laboratory, where he researches the formal foundations of computer security, including covert communication channels, information theory, automata theory, and special function techniques. His current research interests include steganography, insecurity flow, and the inference problem in database design. Moskowitz received a BS and a PhD, both in mathematics and both from the State University of New York at Stony Brook. He is a member of the IEEE and of Sigma Xi.

Contact Myong Kang at Naval Research Laboratory, Code 5540, Washington, DC 20375; mkang@itd.nrl.navy.mil.